Intelligent Decision Technologies 5 (2011) 253–271
DOI 10.3233/IDT-2011-0110
IOS Press

Simulation of autonomous crowd behaviour on Xbox 360

Matthew Brittain[a] and Minhua Ma[b,*]
[a]*School of Computing and Mathmetics, University of Derby, Kedleston Road, Derby, DE22 1GB, UK*
[b]*Digital Design Studio, The Glasgow School of Art, The Hub, Pacific Quay, Glasgow, G51 1EA, UK*

Abstract. A crowd simulator which creates autonomous characters' behaviour in crowds consists many components such as pathfinding, collision avoidance, character creation, behaviour system, and level of details. The majority of these involve different level of decision making in order to simulate autonomous agents' behaviour. Some components have a few different algorithms that can be adopted. For a simulator with a large number of autonomous agents, these components need to be efficient to contribute to the creation of a faster and cheaper game environment. Otherwise bottlenecks may occur and this can led to a poor representation.

In this paper we investigate these areas, discuss and compare existing approaches in each component, and select the best combination on Xbox 360 through a series of experiments on our crowd simulator within the Microsoft XNA framework. We used the Xbox 360 console for accurate testing which is not affected by other processes running in the background. We also optimise the application to overcome bottleneck issues. Our simulator is able to handle a large number of automonous agents with a healthy frame rate of 60 FPS. Based on our implementation and testing results, some recommendations are provided in this paper, which will be useful for independent game developers who create games containing autonomous crowd for Xbox 360 using XNA framework.

Keywords: Video games, crowd simulation, autonomous agents, pathfinding, collision avoidance, level of details, character animation, Xbox 360

1. Introduction

Over the last decade the processing power and memory on computers and consoles has dramatically increased. This increase in performance has led to many developers creating real-time simulators. Crowd simulation is a major research area in video game development over the past few years. It is a process of simulating the movement of a large number of autonomous agents around a scene. A crowd simulator can give a game a more realistic look and feel of non-player characters. Games set in city environments, such as Grand Theft Auto 4, requires the crowd to give life to the game. Crowd simulators are not just for use in video games, they can be used as a cheap form of testing environments, such as building plans, and have been used for management of emergency response, concerts, sporting events, and religious ceremonies. They are faster and easier to identify where the bottlenecks occur and how they can be amended. Every condition can be tested with a simulator and some form of behaviour system for the crowd agents. The number of agents in a crowd has been increasing to a point where full cities can be populated with only a few simple commands.

Crowd simulation is made up of many different components, including pathfinding, collision avoidance, character creation, crowd behaviours, and level of details. Pathfinding is a key part of crowd simulators. Each agent in the scene has a target position. Path finding determines which route the agent should travel to get to this position. There are many different algorithms that can be used to determine this route. Its significance to the simulation is to give a realistic movement of agents around the scene. Collision avoidance is used to prevent collision between other agents and scene ob-

*Corresponding author. E-mail: m.ma@gsa.ac.uk.

ISSN 1872-4981/11/$27.50

jects. There have been much work focusing on each individual component of crowd simulation, e.g. interaction between multiple instances of same character in a crowd, decision-making based on agents' perception and cognitive architecture, and believable behaviour of embodied agents [11], but very few has integrated all these components into a single system. We investigate each of these areas, compare and select the best performing algorithms, and create a crowd simulator on Xbox 360 which is able to handle a large number of automonous agents with a healthy frame rate (60 FPS).

This paper considers selection of AI approaches in crowd simulation to achieve optimal performance and presents our testing results on the Xbox platform. We developed a crowd simulator in Microsoft XNA Framework and run all tests on the Xbox 360 console. Microsoft XNA Framework is a games development environment based on Microsoft .NET Framework 2.0. It allows programmers to create 2D and 3D games for Windows-based PC, Xbox 360 platform, and Windows Phone. XNA framework simplifies the games development process by providing high level C# libraries to manage many tedious and generic technical tasks and allowing developers to focus on game design.

To make sure that our testing is accurate, all projects are created on a standard framework which includes a logging system for recording the game statistics. By testing the applications of various algorithms on a fixed platform, it provides accurate results and from this the best-performing algorithm is chosen. Each test is conducted a number of times and the average result is used to determine which algorithm will be selected. If the projects were tested on a Windows PC, other processes would be running which could interfere the overall testing results. Testing on the Xbox 360 provides uniform testing.

2. Crowd behaviours

A behaviour system controls autonomous agents in a scene. This includes how the agent moves around. One of the most popular behaviour models was created by Reynolds [14]. He created a simulator called "Boids", which simulated bird flocks or fish schools. He summarised 3 basic rules which created a simple steering behaviour for agents:

The separation rule is steering to avoid local flockmates.

The alignment rule is to steer towards the average heading of the local flockmates.

The cohesion rule is to move towards the average position of the local flockmates.

Each individual boid in the flock has to obey these rules. The product of each rule is added together and this forms a force vector. Each Boid in the scene has a unique force vector which is calculated every frame. This method could handle a large number of interacting agents.

Many crowd simulators implemented Reynolds' three rules into their autonomous agent movement and then added more complex rules on top of the base rules. The additional rules can include pathfinding, collision avoidance, and target aiming. The product of all the rules forms a vector to force the agent moving towards a position in the scene. The agents can also move around using steering behaviours such as wander, flee, seek, arrive, or pursuit [13].

One problem encountered using these rules were getting neighbour information. Granberg [7] describes how he used a KD-tree to simplify this problem. His method reduces the number of agents to be sampled by only testing against nearby ones. This makes the calculation faster. It would be the best approach when using a large scene and a large number of agents.

Berggren [2] describes a crowd simulator that he created using Unreal game engine. He managed to get 50 agents in the scene using UT2004 to achieve a healthy frame rate. He suggested that it is possible to increase the number of agents to the maximum 70, but this number can change depending on the type of simulation. This work was published in 2005 and since then we have entered into the next generation of consoles and improvements to PC hardware.

Brooks [3] describes how he created a behaviour system for an AI robot using a layered control system and finite state machines to determine what it should do, like pick a cup up. The available technologies in 1985 made his system very slow. The robot he built could not handle change to state, so changing the environment like putting an obstacle down, meant that the robot did not have vision. With the technology available today, the agents in our simulator always keep checking the environment for change. State machines are used to keep track of what state an agent is in. They are used to make it easier to add new features and stop the code becoming messed up and disorganised.

Sung et al. [16] describes how they studied how real crowds work and what actions influence them, like waiting for a bus. They suggest that the crowd agents be short-term goal based. Based on their research they created a state machine that changes the behaviour of

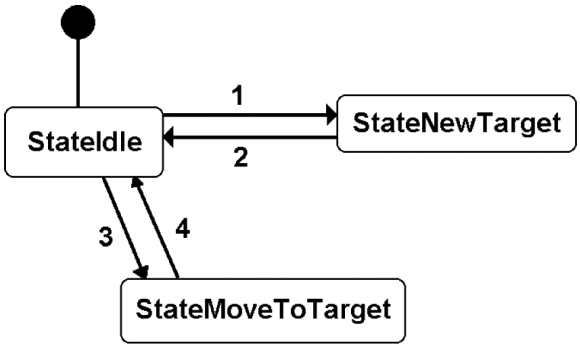

Fig. 1. The state machine used for agents in our simulation.

the agents. Based on this, they created an agent behaviour system that had two goals: a short-term and an overall goal. An example of this is an agent walks to crossing, waits to cross road, crosses road and moves towards the goal.

In crowd simulation, the boids are also known as agents. These agents still use the rules to move around the scene but the agents are more intelligent than the Boids [7]. The agents may have their own state machine and each agent may be in different states. The state machine would select which animation the agent is currently using. The agents could use a pathfinding algorithm to plot its route from its current position to its goal position.

2.1. Finite state machine

The finite state machine (FSM) we use for the project is based on Buckland [4], which is an object based state machine. This type of state machine is a quick and easy to create and debug. State transition is easy to program, and requires only a little computational overhead. Figure 1 shows the state machine of the agents in our simulator.

1. Has no route between the current position and the target, or needs a new target position.
2. Has got a route between the current position and target position.
3. Has got a route to target position.
4. Has arrived at the target position.

The base state contains 3 abstract methods, enter, exit and execute. Each method requires a reference to the agent to be passed in. All states inherit from this class and must contain methods for enter, exit and execute.

The "StateIdle" is the initial state that every newly created agent is placed in. In this state the agent will be playing one of the three idle animations. This state is an intermediate state between getting a new target and route, and moving along the route. If the agent does not have a route to its target, or if the agent has just completed it last route, the agent will change the state to "StateNewTarget". If the agent does have a route to its target the agent will change the state to "StateMoveToTarget".

The "StateMoveToTarget" is the movement state, in this state the agent will travel along the pathfinding route towards the target position. Animation selection in this state is done by comparing the distance between the agent and its main target. The agent contains a float variable to state how far from the target, the agent should be playing the walk animation. If the distance to the target is greater than this value, the agent will play the run animation. The speed the agent travels at is also done in this check. The agent also stores floats variables for max speeds for walking and running. When the agent arrives at their main target, the agent will change its current state to "StateIdle".

The "StateNewTarget" is where a new target is chosen for the agent, and where the pathfinding algorithm is run to obtain the route the agent will follow. As the pathfinding engine can only handle one search at a time, the agent will stay in this state until it has successfully run the pathfinding algorithm. In this state the agent will be playing the jump animation and will remain in the position of its previous target. When the agent has obtained the route to its new target, the agent will change its current state to "StateIdle".

We create this simple state machine which is easy to manage. Granberg [6] describes that "what works for few units may not always work for many", so in our state machine the state transitions are performed with simple calculations, such as truth tables. The transitions are an important part of any state machine, as this is where the problems can occur. A disadvantage to using an FSM is larger systems can be difficult to manage and maintain without a well thought out design.

2.2. Behaviour system conclusion

The agents in our crowd simulator are built based on Reynolds' [14] Boid rules. These rules prevent all agents in the scene from colliding with each other. An additional rule is added, so that the agent will follow the pathfinding route. Each state controls which animation the agent is playing. All agents will move towards a target position and then wait there until a new target position and pathfinding route have been generated.

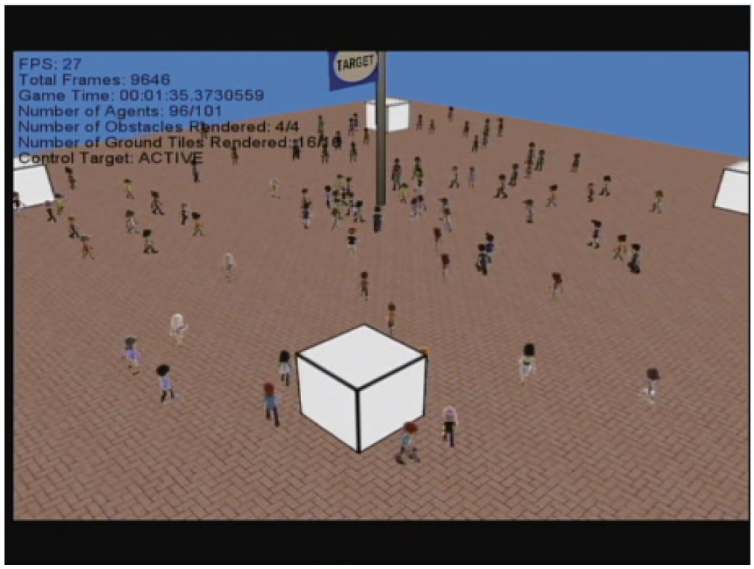

Fig. 2. A screenshot of our crowd simulation.

The application allows the uses of a controllable target. When activated in the scene, all agents regardless of current state will move towards this position. Once the target has been activated it can not be moved, but it can be removed from the scene. Constant movement of the target will cause problems with the pathfinding system. Agents treat this target as their new target position and will generate a path to it using the pathfinding system. Once an agent gets to this position they will revert back to normal movement, create a new target position and move towards that. Figure 2 is a screenshot of the crowd simulator that we created.

One problem using these rules is the obtaining the neighbours information. In a scene with a large number of agents, each agent would need to be tested against. This is a wasteful process and a scene partitioning system would be the best solution to speed up this process. In our project we test against all the agents, and then test against only the agent in the same scene node. This will reduce the number of tests needed and should improve the frame rate. An addition step is required to sort the agents into these scene nodes, but this can also help to improve frame rate. If the scene node is culled, all the agents that reside in that scene node do not need to be rendered, and this would reduce the amount of work that the renderer needs to do.

3. Pathfinding

There are many various pathfinding algorithms which work in different ways to obtain the route between two points. The algorithms run on a graph structure. This graph contains nodes which represent positions. Edges are the connections between each node. The selection of the next node to be used is based on conditions set in the pathfinding algorithms. Each algorithm continues to search this graph until it has discovered the target position, or until all nodes have been visited. We re-create the well-known algorithms, for example: Breadth First Search (BFS), Depth First Search (DFS), and A* search, in the XNA framework. As we want our crowd simulator to have goals or target positions for each agent, finding the path needs to be done quickly but may not be the quickest.

3.1. Pathfinding algorithm

When looking for a path, the first path to be discovered may not be the shortest. For a crowd simulator, the cost of searching needs to be as cheap as possible, as each agent in the scene will be doing this.

Breadth first search works by using an open and closed list. Once all nodes have been visited the algorithm stops. This will tell the agent if a path is possible between the start and goal node, but the resulting path is the first one to be discovered. This may not be the quickest though. This search algorithms uses a queue and the first-in first-out approach. This algorithm searches the entire graph, but can be closed once a route is discovered. This algorithm offers no advantages by searching the entire graph once a route

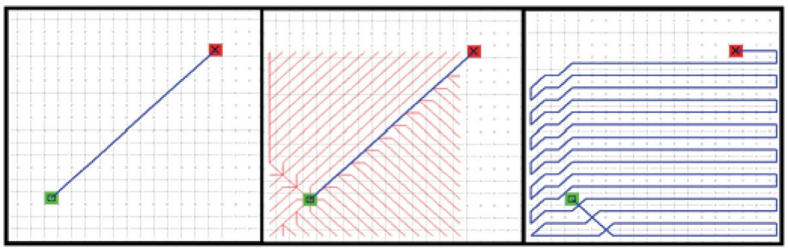

Fig. 3. A*, BFS and DFS routes between 2 points.

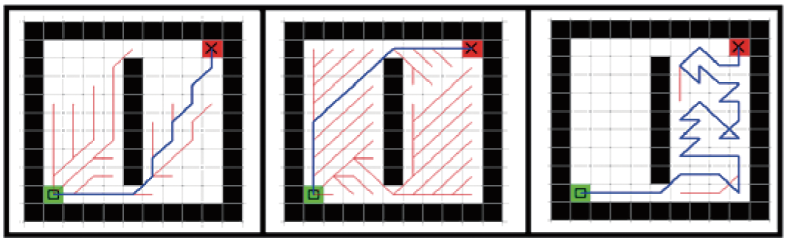

Fig. 4. A*, BFS and DFS routes between 2 points in a scene with obstacles.

is discovered. Once the finish node has been visited, it can not be changed if a smaller route is discovered.

Depth first search is similar to breath first search, but uses a stack instead of a queue. Instead of first-in first-out, its approach is first-in last-out. This works the same as breadth first search as when the finish node is discovered it can not be re-visited again.

Dijkstra's algorithm solves the shortest path problem and find the shortest path to the target node. However, it is wasteful as it stores the path to every node and then throws away all the paths that are longer than the shortest [13]. This algorithm may not work well in a crowd simulator, due to the time and cost of generating a path for every agent in the scene.

A* search was first described by Hart et al. [9]. It is simple to implement and is very efficient. The algorithm requires a heuristic estimate to be calculated for node evaluation. A* is similar to breadth first search in the way that it will always find a route if there is one.

Sornum et al. [15] describes creating a subway station simulator using a collision map. It simulates an explosion in part of the station to test where bottlenecks can occur in the station. They tested different game engines with 100 agents and 1000 agents. From their findings they managed to get 15 frames per second with 1000 agents and 175 frames with 100 agents. Their paper shows how a 3D environment can use a 2D collision map to keep track of agents.

We use Euclidean distance, which is the length of the vector between the node and the route target position, as heuristics in A* search. This requires that all nodes in the graph are positioned in the correct position. A problem with this approach is that the heuristic is only an estimated value. The value that the heuristic calculation gives could be going through an obstacle. The next node selected is based from this value. This would make the time taken to find the route longer.

The BFS and DFS are basically the same, but a BFS uses a queue and a DFS uses a stack to store nodes in. The graph has vertical and horizontal edge connections, which have a cost of 10. Diagonal connections can be switched off, but if they are used will have a cost of 14.

Figure 3 shows the route that each algorithm takes to get from the start node to the target node. On the image, the blue line represents the route taken and the red lines show node connections evaluated. The A* search went straight towards the target without testing any other nodes. The BFS algorithm searched in all directions from the start node, not revisiting any that had been visited before. The overall route was the same as the A* search algorithm, but took longer and visited lots more nodes. The DFS algorithm started by heading away from the target, and its route nearly covered the entire graph; this was very wasteful for a simple graph that contains no obstacles.

Figure 4 shows a scene with obstacles blocking a direct path between the start and target nodes. In this case all 3 algorithms had a different route to the target. In this graph the A* did evaluate other nodes. The BFS algorithm tested every node in the graph before finding the target, but its route cost was equal to that achieved by the A* algorithm. The DFS algorithm did find the

Table 1
10 × 10 graph – pathfinding results

10 × 10 Graph	A*		BFS		DFS	
Diagonal Connection	Yes	No	Yes	No	Yes	No
Average Time	0.00060804	0.00055913	0.00063599	0.00088489	0.00063521	0.00062309
Minimum Time	0.0005296	0.0005505	0.0005889	0.0005578	0.0006124	0.0005823
Maximum Time	0.0006203	0.0005921	0.0006999	0.003471	0.000721	0.0006706
Route Cost	126	180	126	180	126	540

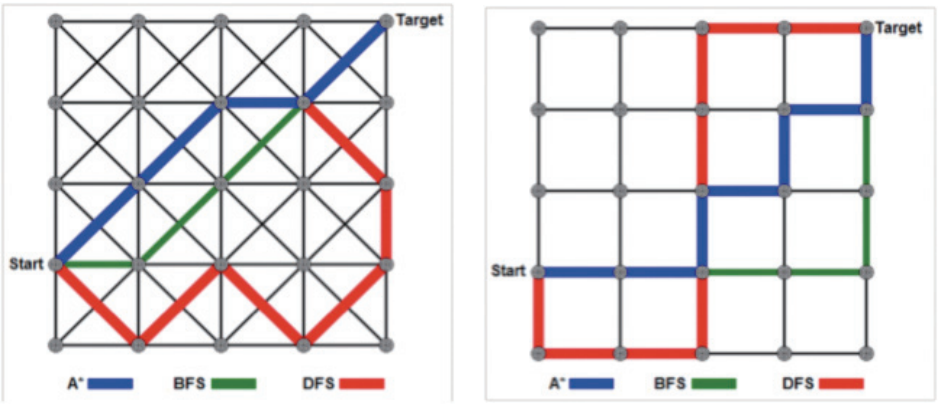

Fig. 5. All algorithms running on a 5 × 5 graph with diagonal connections on (left) and off (right).

target node, but the route it took went through all the nodes on the right side of the graph, this was wasteful.

3.2. *Pathfinding testing*

For the testing of the pathfinding algorithms, we create a project which displayed the route each algorithm took. This application allowed the grid size to be changed, remove some nodes from the graph and switched between graphs with, and without diagonal connections. This application rendered the route each algorithm took to reach the target. Figure 4 left figure shows the application displaying the route taken by each algorithm on a 5 × 5 graph that had diagonal connections. The route for each algorithm was different, but A* and BFS did have routes which had the same route cost. The DFS algorithm took a longer route to the target. The problem was that because the DFS algorithm never came across a node that does not have any other connections, its current branch would not end until it reached the target. Figyre 4 right figure shows the same test conducted on the graph, without diagonal connections. On this graph all the algorithms had a different route, but A* and BFS again had routes with the same cost.

For the testing of the algorithms, we created a facade application that just displayed the time and cost each algorithm took to find the route. Rendering the graph alone took up a lot of processing power and only produced a low frame rate. The tests were conducted on graph sizes of, 10 × 10, 25 × 25, 50 × 50 and 100 × 100. The application was built on top of the base framework that we had created. All tests used the same start and target position and each test was conducted 10 times. Each algorithm was tested with diagonal connections on and off.

In the 10 × 10 graph, there are a total of 100 nodes. The number of edges for this graph with diagonal connections is 342 and without diagonal connections there are 180. Results summary below (Table 1). All tests on a 10 × 10 graph with diagonal connections on resulted in the same route cost of 126. With diagonal connections off, the DFS algorithm route cost was triple that of both A* and BFS. In the results of the time taken to find the route, A* was the quickest, making this the overall winner of this graph size.

On a 25 × 25 graph, there are a total of 625 nodes. The number of edges for this graph with diagonal connections is 2352 and without diagonal connections there are 1200. Results summary below (Table 2). All tests on a graph with diagonal connections on resulted in the same route cost of 336. With diagonal connections off, the DFS algorithm route cost was seven times that of both A* and BFS. In the results of the time taken

Table 2
25 × 25 graph – pathfinding results

25 × 25 Graph	A*		BFS		DFS	
Diagonal Connections	Yes	No	Yes	No	Yes	No
Average Time	0.00056102	0.00057297	0.00060251	0.00059057	0.00143287	0.0006303
Minimum Time	0.0005336	0.0005511	0.0005511	0.0005554	0.0006088	0.0005946
Maximum Time	0.0006218	0.0006542	0.0007019	0.0006588	0.0048903	0.0006936
Route Cost	336	480	336	480	336	3360

Table 3
50 × 50 graph – pathfinding results

50 × 50 Graph	A*		BFS		DFS	
Diagonal Connections	Yes	No	Yes	No	Yes	No
Average Time	0.00059284	0.00056588	0.00064481	0.00060206	0.00063776	0.00063527
Minimum Time	0.0005293	0.0005437	0.000591	0.0005595	0.0005634	0.00059
Maximum Time	0.000806	0.0006743	0.0007855	0.0007604	0.0008129	0.0006929
Route Cost	686	980	686	980	686	12740

Table 4
100 × 100 graph – pathfinding results

100 × 100 Graph	A*		BFS		DFS	
Diagonal Connections	Yes	No	Yes	No	Yes	No
Average Time	0.00055806	0.00056243	0.00059683	0.0005903	0.00062093	0.00064572
Minimum Time	0.000552	0.0005283	0.0005683	0.000556	0.0005916	0.0005991
Maximum Time	0.0005675	0.0006352	0.0006536	0.0006026	0.000636	0.0007291
Route Cost	1386	1980	1386	1980	1386	49500

Table 5
A* search results summary

Graph Size	Diagonal Connections	Average Time	Cost
10 × 10	ON	0.00060804	126
10 × 10	OFF	0.00055913	180
25 × 25	ON	0.00056102	336
25 × 25	OFF	0.00057297	480
50 × 50	ON	0.00059284	686
50 × 50	OFF	0.00056588	980
100 × 100	ON	0.00055806	1386
100 × 100	OFF	0.00056243	1980

to find the route, A* was the quickest, making this the overall winner of this graph size.

On a 50 × 50 graph, there are a total of 2,500 nodes. The number of edges for this graph with diagonal connections is 9,702 and without diagonal connections there are 4,900. The data collected from this test can be found in Table 3. All tests on a graph with diagonal connections on resulted in the same route cost of 686. With diagonal connections off, the DFS algorithm route cost was thirteen times that of both A* and BFS. In the results of the time taken to find the route, A* was the quickest, making this the overall winner of this graph size.

We repeat this experiment on a 100 × 100 graph, which has 10,000 nodes. The number of edges for this graph with diagonal connections is 39,402 and without diagonal connections there are 19,800. The data collected from this test can be found in Table 4. All tests on a graph with diagonal connections on resulted in the same route cost of 1,386. With diagonal connections off, the DFS algorithm route cost was 25 times that of both A* and BFS. In the results of the time taken to find the route, A* is the quickest algorithm again.

3.3. Pathfinding conclusion

The A* search algorithm was the best performing algorithm on all graph sizes. It produced the quickest search times, and the smallest route cost each time. The BFS algorithm did match the A* search route cost, but was slower. The aim of the testing is to discover the algorithm which had the quickest search time and route cost on Xbox. The A* search algorithm is the best from the tests we conducte and this will be the algorithm used in our crowd simulator. A* search results summary below (Table 5).

On the 100 × 100 graph with diagonal connections on, the average time taken to find the route was the quickest of all graph sizes. This was surprising since this graph had 10000 nodes. All tests were conducted under the same testing environment, so this value is accurate. But due to the amount of memory this graph would require, the final version should be as small as

possible. A graph similar to the 25 × 25 graph would be the best option, as this was the graph which had the second quickest average search time. This graph size would also offer a greater number of paths for a scene with lots of agents.

For our crowd simulator, we adopt Tecchia et al. [17] collision map system for generating the pathfinding graph. This would be a texture with the width and height used to set the size of the graph. Each pixel represents a node in the graph, and pixels of a certain colour could be removed. The advantage of this approach is the pathfinding graph is not hardcoded, making it easier to change for different testing environments.

4. Collision avoidance

Collision avoidance is a system that prevents collision between two objects by changing the course to avoid the collision. This concept is used in aviation in a system called TCAS (Traffic Collision Avoidance System). The system monitors the airspace around an aircraft for other aircraft equipped with a corresponding active transponder, independent of air traffic control, and alerts pilots if there is a threat of midair collision and tells each plane to either rise or drop in attitude.

Using a collision map or a grid based map [17] is a popular approach of collision avoidance. The collision map would be best suited for scene objects, i.e. the objects in the scene which do not move. They are simple to implement, but problems can occur if a group of agents surrounds one another and there is no place to move to [12]. One solution to stop this problem from happening is to send out 2 rays from the agent [7]: one straight ahead and the other 90 degrees to the left. If these rays intersect with another agent then move the agent to the right, away from the other agents. This method also works with scene obstacles. The demo provided by Granberg uses a cylinder to represent each agent. Then each ray is checked against these cylinders. If the radius of cylinder multiplied by 3 is less than the magnitude vector of the agent to the obstacle. Then move the agent away from the obstacle. Otherwise no changes are to be made.

Granberg [6] describes his method for working out the cost of a route in terrain based environments. The environments have multiple height levels and need to be special cases when working out the cost of the path. When creating a node map for terrain it is best to first split the terrain in sub-terrains. Then treat these sub-terrains as a node. Van den Hurk and Watson [19] proposed a multilayer flocking system which simulates the movement of crowds containing characters of vastly different sizes and allows agents to move underneath other agents when there is sufficient space to do so.

Reynolds' [14] behaviour rules stop collision between other agents. The first rule is separation which is to steer to avoid crowding local flock mates. As long as these rules are applied to each agent in the scene, there should only then be checks for collision with scene obstacles. It helps to split the scene up into KD-trees, a method described by Granberg [7]. This way only an obstacle in the agents' sector will be tested against. This saves processing power and improving frame rate as fewer checks are needed. An addition rule for collision avoidance can be added to handle obstacle avoidance. Granberg [7] created a crowd simulator which only used cylinder obstacles to test collision avoidance. In his example each obstacle was tested against each obstacle and if the agent was in proximity to it, the obstacle would apply a force to push the agent away from it.

4.1. Priority system for agents

When a pair of agents are tested and is discovered to be on a collision course, one of the agents must make an adjustment to his route [5], their collision avoidance algorithm uses a priority system for deciding which agent makes an adjustment to their route. A variable is stored on the agent to represent the priority of that agent. This could be an integer number between 0 representing low priority and 10 representing high priority. The agent with the lowest priority in the pair will be the agent that will adjust his route. If both agents have the same priority, then one of the agents is randomly chosen to have their route adjusted.

Before the agent priority factor comes into effect the first calculation is to predict if the agents are on a collision course. By testing this first, it allows more expensive calculations to be avoided, freeing up more CPU cycles which will help to improve the frame rate.

Using the priority system for the agents will mean only one agent will need to adjust their route. The adjustments that the agent can do are the following:

- Change direction only. The agent will change its direction to avoid collision with the other agent.
- Change speed only. The agent can slow down, so that the other agent can get to the intersection point first, or it can speed up so that this agent can get to the intersection point first.

– Change direction and speed. Combination of the above statements. In our simulator we use this method. We created a sub-target which is on the vector to the target from the agent current position and then offset this by the collision avoidance radius.

4.2. Linear equation

The equation of a linear line $y = mx + c$ can be used to estimate the position at anytime if the values for m and c are known. This is used in our crowd simulation to estimate the position of an agent and test if the agents' current vector is on a collision course.

Each agent has a position and a velocity vector. The 'm' variable can be discovered by dividing the Y (in 2D) or Z (in 3D) by the X value of the velocity vector. This will produce the 'm' variable value. The 'c' constant value is discovered by re-arranging the "$y = mx + c$" formula, this becomes "$c = y - mx$". With the 'm' variable value already discovered, we replace the 'x' and 'y' with the position vector values for X and Y (or Z in 3D). Now 'c' can be worked out by calculating the right hand side of the equation. At this point the 'm' and 'c' values are now known for the agent. Future positions can be discovered by putting a value into the 'x' value in the formula.

Collision detection can be calculated between two agents by using a simultaneous equation. If each agent has a linear equation, the simultaneous equation will show where the agents would collide. As the position may not be in the current scene, a check would be needed to test that the intersection point is in the current scene. The intersection point can be in a location that the agent has already passed, so a check would be needed for this as well. This system may not work well in a scene with a large number of agents. It would be most likely that an agent would be on a collision course with lots of other agents. The intersection point may be in a remote location to where the agent currently is. There would need to be a system to control how far an agent must be from this point for any changes to happen. This algorithm requires a lot of expensive calculations to be performed and may not be the best option to use.

4.3. Obstacle force

Granberg [7] created an addition rule for collision avoidance that worked with Reynolds [14] Boids rules. The algorithm checks agents against all obstacles in the scene and any obstacles that are near an agent return a force to push him away. The system only used cylinder obstacles, probably due to the fact that they offer a better performance than bounding boxes. In our project we use box obstacles to represent buildings. We create box obstacles which use the same code as cylinders, but increase the size of the radius to enclose the entire box shape. One problem of this is low fill-efficiency, or a cube obstacle would need to be made up of multiple cylinders.

4.4. Optimizing collision avoidance check

Some collision avoidance algorithms such as Foudil et al. [5], checks every pair of agents. The main disadvantage to this method is testing against every single pair of agents. In a scene with only a few agents this check is acceptable, but in a scene with 50 agents, it would require 1225 tests, and 100 agents would require 4950 tests. Performing this numbers of tests each frame could lead to a poor frame rate and bottlenecking.

One way to reduce the number of tests is to use a scene partitioning system. This would split the agents into different scene nodes with only the agents in each node being tested against. If each scene node had 10 agents in it and there are 8 nodes, this would mean only 45 tests per scene node and a total number of tests being 360. If all the 80 agents in the scene were tested against each other there would be 3160 test every frame. Using a partitioning system saves 2800 tests being performed.

In our crowd simulator we test against all agents in the stage one to see how this affects the frame rate. In stage two we only test against agents in his current scene node. This should improve frame rate as less checks are required, but a sorting system will be needed to sort the agents into scene nodes.

4.5. Projects created

We create a project based on Duthen et al. [5] priority system algorithm. In the test project we use 2 agents that travel towards each other. We added a ring around each agent to show where the detection limit is and a rectangle to show the direction the agent is travelling in Fig. 4. It shows the stages that happened when the agents approached each other. In the first stage the agents are approaching each other, but are currently not in detection range. In the second stage the agents are in detection range, and are on a collision course. The left agent has a lower priority than the right agent, so this is the agent that changes it course. A new sub-

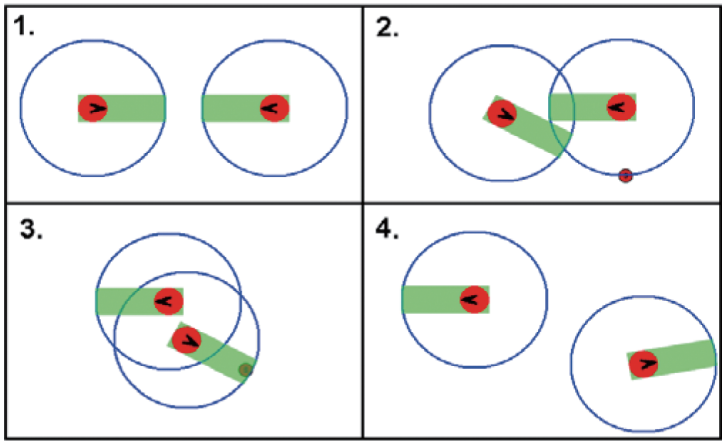

Fig. 6. Priority system, stages taken by the algorithm to avoid collision between agents.

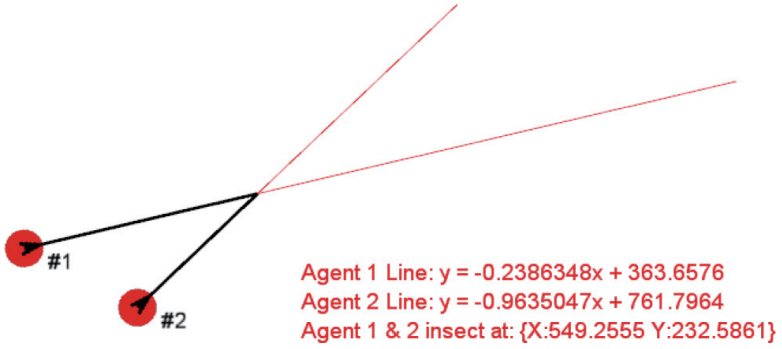

Fig. 7. Agents using a simultaneous equation to discover where they would collide.

target is created halfway between the current position and target position and to the side of the second agent. The third stage shows the agents passing each other, not colliding. In the fourth stage, the agent that headed to the sub-target has reached it and is now proceeding to its original target.

In this case the collision avoidance system worked as intended. The agent with the lowest priority moved out of the way to avoid colliding with the other agent. But in a system with a larger number of agents, this algorithm may not work well. The sub-target generator may cause an agent to go too far off course. This system worked well, but the behaviour system using the separation rule would handle this anyway and be less CPU expensive. This system did not handle collision avoidance with obstacles that well. In some cases the agent would travel towards an obstacle when the sub-target was generated. This algorithm is best suited for agent checks against other agents and not obstacles.

In the project created based on linear equations, the agents would find the position where they were going to collide at, but in some cases this position was too far in the distance, or the position was behind the agent as it was moving away from it. This project was to test where a set of agents would collide (Fig. 6). This could be used in the priority algorithm as this provided a faster method of testing if a set of agents would ever collide with each other.

In the project created based on Granberg's [7] example, the collision avoidance was for use with the scene obstacles. The agents each used a behaviour system that was implemented based on Reynolds [14] Boids rules, this stops the agents colliding with each other. A change we made to Granberg's example was to use box obstacles, instead of cylinders. The boxes had a cylinder that enclosed all of it, and from the example this worked very well (Fig. 7). The agents would avoid the obstacles and move around the scene without any problems. This algorithm was the easiest one to integrate and worked with the behaviour system as an additional rule.

Through our experiments, we find that Granberg's [7] example of using a force on each obstacle is best suited

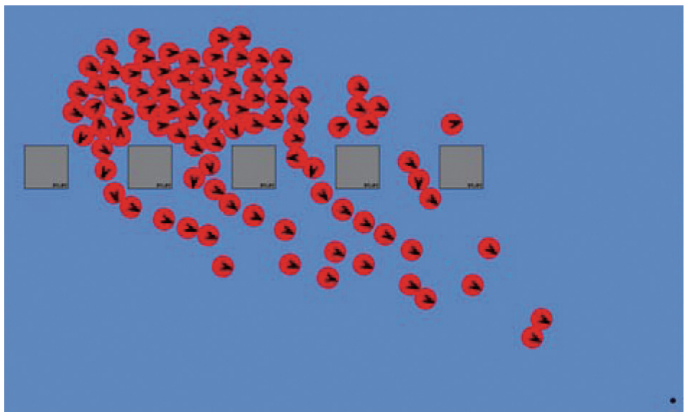

Fig. 8. Collision avoidance using an obstacle force.

for the collision avoidance system. The system was the easiest one to integrate and did not require lots of expensive checks. The linear equation approach needed to be updated every frame, as the agent would decrease and increase in velocity and the simultaneous equations would find an intersection point too far in the distance. Duthen et al. [5] priority system worked well with the lower priority agent moving out the way but this seemed redundant as the behaviour system would handle this anyway.

5. Character creation

Character creation is done in many ways. Some methods are simple having a mesh per character, but this can be very repetitive. Games like GTA, Prototype, and Left 4 Dead for example, use this method.

To get a larger number of agents in the scene at the same time it is best to use different resolutions of models. Granberg [7] describes his method of using higher polygon meshes for agents near to the camera. Then fewer polygons depending on the distance from the camera. This method is called level-of-detail (LOD). He suggested that low resolution models should not have animations, shadows and collisions. Medium models can have animations without blending or callbacks, but they should have some collisions detection and shadows. High resolution models which are near to the camera should display all character features.

Therien et al. [18] explained how they did the crowd system in Assassin's Creed. They only updated the agents that are currently visible on screen. They said that they had to make the crowd as cheap as possible. They had no level of detail on the behaviour system of the NPCs in the crowd. From this we would follow their method and only update the agents currently on screen. We also look at how a level of detail system could work on the behaviour system we developed.

Character creation is the process of creating the character assets used in the application. The application we create uses the Xbox 360 avatars. Using the avatars in application was introduced to the XNA framework for Xbox 360 projects in XNA 3.1 released in June 2009. The Xbox 360 avatars are animated using skeletal animations. The heads are an animated by changing the textures for the eyes and mouth. The avatars come with a selection of default animations that include waving, clapping, standing idle and celebrating. These default animations do not include movement animations such as walking or running. The XNA creators club does provide a sample that includes loading and playing back custom animations.

Figure 8 shows the avatar rendering stages. The left image is the avatar in the bind pose, the default positions for all vertexes in the avatar model. The next image is the skeleton animation data; this is a list of transforms for each bone in the avatar model. Both are passed into the GPU where the bind pose vertexes are transformed by the skeleton animation data to form the avatar, the result is shown in the last (4th) image (the 3rd image is the resulting avatar rendered in wireframe).

5.1. Custom animations

The custom animations require the XNA framework content pipeline to load. The sample provided by the XNA creators club requires three projects to be included to the solution. The animation assets require the content processor to be set to this new content pipeline.

The files for the custom animations are over 10 MB each. This is because the avatar is made up of 71 joints,

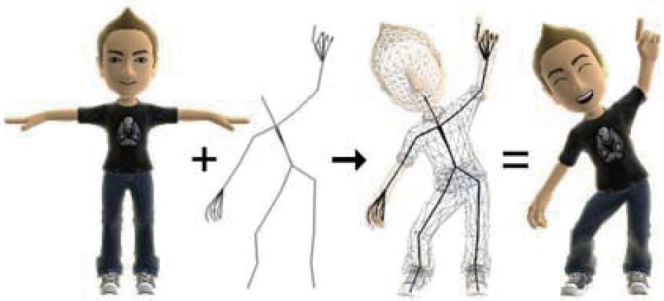

Fig. 9. The avatar in the bind position with the animation data and combined together.

with half taken up by the hands alone. The character animation in our application does not involve hand movements, but the avatars still need these transforms. The rendering of the avatar model is handled in the XNA framework. This area cannot be modified, so optimizations, such as using discrete level of articulation (the idea is similar to discrete LOD, but applys to number of joints), cannot be done on this.

5.2. Animation manager

As the custom animations are large files, having multiple versions of each will use too much memory. To overcome this we have created an animation manager class. This class is a singleton class which means only one instance can exist at any time. A singleton class provides a global access point to get the data stored inside it. The animation manager class we have created loads all the custom animations and default animations used by the agents in the simulator. All animations are updated once in this class every frame. The animations are only updated once to save on processing time. This gives a better frame rate, but because the animations are only updated once, agents may appear to jump from one animation to another. When an avatar is rendered, the bone matrix transforms will be retrieved from this class for the animation that the agent is currently playing. These are in the form of a matrix list. This list is passed into the avatar render function, the bind pose transformed are transformed by the matrix list and the result is the position that the avatar is rendered in.

There are a total of 7 animations that an agent can play. 4 are default and 3 are custom, they are:

- Idle1 – Default
- Idle2 – Default
- Idle3 – Default
- Wave – Default
- Walk – Custom
- Run – Custom
- Jump – Custom

Granberg [7] created a DirectX/ C++ system that used a face factory. In this factory he would load an .X file that contained different face models. A random face would be generating from all the models using vertex blending. As the same mesh for the body is used for all characters and only the face model would be different, he would render the body mesh and put in a if statement to check for when the bone being rendered was called "Face", when this bone was found it would render the face model from the face factory. Using this method would be the best way if all characters in the scene need to have the same body mesh, such as soldiers, but this does not give a wider range of different appearances for the agents.

The XNA example is from the XNA creator's club website [17]. The sample uses the XNA framework content pipeline to load and play back the custom animations. The sample only supports one avatar in the scene, but does contain code for creating a random avatar or using the users own profile avatar. The sample only supports the playback of one animation at a time, but this can be extended to do animation blending.

5.3. Character creation testing

In the testing we created a new avatar instance for each agent in the test. So in a test with 50 agents, 50 different agents needed to be created and stored in memory. With 50 agents in the scene the best frame rate achieved was 32 frames per second. Each test would produce an average frame rate for the first 30 seconds of the test. This is the number recorded from running each test. All avatars used in the tests were randomly generated at run-time. Some avatars had extra items on them, such as accessories (glasses and hats). These are added items on the top of the base avatar model.

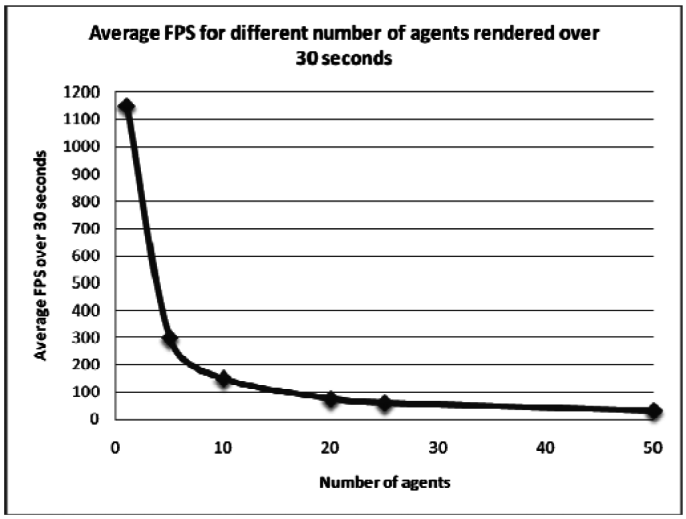

Fig. 10. The average frame rate for rendering different numbers of Xbox avatars.

Table 6
Results summary for average FPS over 30 seconds

Number of agents	Average FPS over 30 seconds
1	1148.8
5	299.8
10	149.8
20	75.2
25	60.4
50	32

These additional items contained additional polygons that needed to be rendered.

We tested the application with the following numbers of avatars, 1, 5, 10, 20, 25 and 50. Each test was performed five times and then an average frame rate was calculated. Table 6 shows the average results for each number of agents rendered. Figure 9 shows a line graph with the average frame rate for all tests conducted. When a single agent was rendered the average frame rate was near 1150 frames per second, but when 5 agents were rendered, this dropped down to roughly 300 frames per second. The next drop was roughly 50% from five agents (299.8) to 10 agents (149.8). When the 20 agents (75.2) were rendered, this too was roughly a 50% drop as well.

5.4. Character creation conclusion

From the testing we conducted on the character creation project, we discover that to achieve a frame rate of 60 frames per second that only 20–25 agents should be rendered at any time. The frame rate we collected for these tests were just for the character creation section, not including the pathfinding, behaviour or collision avoidance system. The other systems take up process as well and because of this the number of agents needs to be limited. In the testing all the avatars were rendered, even if they were not in the camera view frustum. Frustum culling would increase frame rate, but if all agents were viewable in the scene then this would be an accurate frame rate.

When we tested the application to see how many avatars the scene could handle before crashing, the result was 143. This was in a scene that was just rendering avatars and had no other agent systems. Because of this we have put a maximum limit of 125 agents in the scene at any time on the final version. Berggren [2] suggested that the limit for maximum number of agents should be 70, but this was tested five years ago and since then technology has improved.

The agents in the scenes will be represented by the Xbox avatars. These character models require no stored assets as they are generated at run-time. This means that any number of avatars can be created to give the simulator a greater variation of characters.

As the avatar models use the same skeleton structure, the animations for all types will be the same, saving memory. Each custom animation is over 10 MB in size and if a version for male and female avatars were required, this would require double the amount of memory to be used.

6. Level of details

Level of detail (LOD) is the process of decreasing the amount of polygons rendered that a 3D object has,

as it moves away from the camera. The purpose of an LOD system is to reduce the workload of renderer and hence increase frame rate. If an object is only taking up a few pixels on the screen, it is not necessary to render hundreds of vertices. An LOD system selects how many vertices should be rendered on the model, based on its position relative to the viewpoint. LOD is not only for geometry, but also used in textures, i.e. mipmaps, and joints of articulated characters.

Discrete LOD involves three stages: generation, selection, and switching. Generation is the stage where the different version of the object is created. This can be accomplished either manually or automatically. Manual generation is the preferred method which requires multiple versions of the same asset being created with different polygon counts. The advantage to using this method is that the object will still be how it was intended to be created. The disadvantage to using this method is an artist would need to create multiple versions of the same asset, but with geometry modelling tools such as 3DS Max and Maya, the optimizing can be done with just a few mouse clicks. Automatic generation is where a single version of the asset is loaded, this is the high resolution version which would be used for objects near to the viewpoint. Then an algorithm is run to collapse the vertices down to a level that is suitable for rendering [10].

The selection stage is choosing which version of the object should be rendered. Methods for doing this can included the distance between the object and the viewpoint or counting the number of pixels the object takes up on screen. The second method would be best suited to an application that is also performing occlusion queries. The occlusion query would stop objects being rendered that are behind other, but the cost for this would be a lower frame rate as the scene is rendered twice, once to a texture and then to the screen. There is also a stage where the pixels need to be counted in the texture to determine which version of the object to use.

The distance from object to view point is the most common method and the distance can be used in a range-based selection system to select the version of the object to use. Issues with this method are the point on the object being tested needs to be the closest point to the viewpoint. If the center point is used, this can cause bad selection.

Switching is where one version of the object is switched for another version. The main goal of this is to make it appear as nothing has changed. There are different ways this can be accomplished [1] describes that an abrupt model substitution is often noticeable and distracting, this is called popping. The models should change at distances where the change would be unnoticeable.

Hysteresis is simply a lag between transitions of different LOD version (Luebke et al., 2003). Each object has two different ranges for the LOD selection, one for increasing distance and one for decreasing. So using this method an object will stay in its high version even past the point where it changes from median into high version. A disadvantage to using this is, if you stop on the boundary it will cause the object to flicker between different LOD versions.

6.1. Character LOD

Granberg [7] describes that scene with a large number of animated agents should have three (high, median and low resolution) LODs. Each level has different rendering and animation rules. The high level is for agents near to the viewpoint and low level for the agents in the far distance. The low LOD for agents should not have any animations, shadows and should not perform collision checks. The median LOD can have animations but no blending or callbacks. They are allowed collision detection and shadows. The high LOD should have everything, animations with blending, callbacks, collision detection and shadows.

6.2. Example projects created

The XNA framework currently (version 3.1) offers no control of level of detail for models. This means that models need to be generated manually. For buildings in the scene, we created three versions of the same box shape. Five faces are created, one for each of the four side walls and one for the roof. Each face is created based on a quad shape. The data is obtained from passing in the dimensions of the building when it is being created. The low LOD version has 2 polygons that make up each side (10 polygons total). The median LOD version has 8 polygons (40 polygons total) and the high version has 18 polygons (90 polygons total).

Each building has an update function that requires the camera position to be passed in. The distance is calculated between the viewpoint and building position. Based on this range-based selection, the render function determines which version to render. A problem of this approach occurs when it deals with a very large object. No single LOD can adequately represent both the portions of the object near the camera and the portions distant from the camera. This may cause the object to

use the lower LOD when part of it should be rendered in a better quality. Instead of using a uniform distance for every object in the scene, when the object is created two float values are needed, so that different objects can set their own range distances for its different LOD versions.

The avatar rendering is done in the XNA framework and we do not have access to this code. The renderer might have some form of LOD, but without seeing the internal code, it is unknown. This is one of the drawbacks of using XNA engine. An advantage of using the Xbox avatars is that they all use the same animation data, which means that memory can be saved.

6.3. Testing results

We conducted 10 tests for a scene with LOD settings and a scene without any LOD settings. The results from these test can be found in Table 7 (no LOD settings) and Table 8 (LOD settings). The data collected from these tests, was the average FPS over 30 seconds. The scene without any LOD settings had an average FPS of 387.53, but the scene with the LOD settings had an improved FPS of 400.55.

Table 7
Results of application without any LOD settings

Not using LOD	
Test	Average FPS over 30 seconds
1	388.2667
2	388.6667
3	386.2667
4	388.7667
5	387.0667
6	389.2333
7	384.7333
8	388.1668
9	388.3333
10	385.8
Total Average	387.53

Table 8
Results of application with LOD settings

Using LOD	
Test	Average FPS over 30 seconds
1	400.8
2	400.4667
3	398.8333
4	401.5667
5	401.3667
6	398.8
7	402.2
8	400.6333
9	398.8667
10	401.9667
Total Average	400.55

7. Implementation and testing

To achieve accurate testing results, all projects run on a fixed platform, Microsoft Xbox 360 console. The projects are created using the XNA framework that works with the C#. The framework includes a system for recording the game statistics, which is used to run analysis on each project.

7.1. Specification

Hardware: Xbox 360 console with hard disk drive and valid XNA creator's club membership. There must be a connection to Xbox live to run the application.

Software: Visual Studio 2008, and XNA 3.1

Test platform: Xbox 360 console.

Required features for all tests: Statistics on frame rate.

Screen resolution: 1280 * 720 pixels.

Input: Xbox 360 controller, no keyboard or mouse.

The application is built from a simple framework. This framework was created first and used as the backbone for all applications. The base framework will set the application up to work in 1280*720 resolutions. Only input support for the Xbox 360 control pad will be supported. There will be a debug class that stores statics such as frame rate, current run-time and how many clients are connected to the application. There is also a camera class, which supports moving the camera around the scene.

All tests were performed in the same style. This gives uniform testing results, so that accurate results can be found.

7.2. Testing

All applications are tested using the same test plan and must run under the same testing conditions. To make sure that the tests are accurate, all tests are conducted at least 5 times. The results then be averaged and this is the value which is evaluated.

The following are tested in each case:

- Frames per seconds
- Average frames per seconds

Some combinations of specification will also require extra tests. For example, when testing the path finding algorithms, the amount of time each algorithms takes

to work out the route and the cost of the route. Once all tests have been conducted the necessary amount of times, each will be integrated into the final application.

The final project is the final version of the crowd simulator. It is built from the best performing components found from previous testing. The final project is tested in two stages. Stage 1 is the project which is composed of all the component projects. The project is then optimized and tested in stage 2. Stage 2 testing shows if any bottlenecks have been fixed and how much the frame rate has improved, and whether it gives a better frame rate performance than those of the existing crowd behaviour simulators.

7.3. Implementation

There are different types of behaviours that agents can demonstrate. They can include steering behaviours such as seeking a position or wandering around the scene. The behaviours can also be the current state that the agent is in. This is done in the finite state machine. This state machine decides what the agent should currently doing, e.g. moving towards a goal or looking for something in the scene.

To get the best number of agents in the scene at the same time, there are many rendering options. The scene can be split into a scene graph which stops areas out of the cameras' view from being drawn. The number of polygons in the scene objects can help improve performance as well. A simple building with no detail other than that of a texture can be as little as 10 polygons.

The level of detail of the characters can be reduced depending on the distance of the agent to the camera. If the agents in the scene are using animations then there will be a cost of this. We investigate the different between frame rates for animated and non-animated agents and consider the realism-performance tradeoff.

7.4. The world

The world is built using a texture, based on Sornum et al. [5] collision map, in which black pixels represent the obstacles in the scene. The world dimensions are taken from the texture representing the world. Using this information about the world, the pathfinding graph will be created. First a graph will be created that is the same size (width and height) as the texture. The graph is searched using the A* search algorithm which requires a heuristic value for evaluating the nodes, so the position of the nodes is very important as the Euclidean distance between nodes is used for the heuristic value. The edge connections are then added to the graph, diagonal connections are used. All vertical and horizontal connections have a cost of 10, where diagonal connections have a cost of 14. Once a completed graph has been created, any node which represents an obstacle is removed from the graph. This prevents agents trying to get to an unobtainable position.

The ground tiles are in a 4 by 4 grid (16 in total), each representing a 1/16 of the world. The texture coordinates are scaled so that the bricks in the ground texture remain relative to the size of the Xbox avatar.

A problem with using a texture to create the world is that all content files used by the XNA framework must be an ".xnb" file. This file can only be created with a build. The file itself can only be deployed to the Xbox 360 console; it can not be directly accessed and edited. There has been some plug-ins created that save into this type of file, but there is no way to put this on the Xbox 360 without a deployment, so it is best to just let the build generate this file.

The world has a total of 4 obstacles in it. These are positioned near each of the 4 corners. The camera is kept in the same position for all tests and does not move. All tests were conducted 5 times and the average result is used. Each test was run over 2 minutes and the frame rate was recorded at 5 different intervals (10, 30, 60, 90 and 120 seconds). Each test was tested with a different number of agents (10, 25, 50 and 100). In all tests, the avatars models, starting and target positions were randomly generated.

7.5. Stage 1 testing

In this testing stage, we test just the application which consists of all the test projects. This was to get initial feedback from the application. All but the character creation test projects were created in 2D and this was the first time that all areas were integrated together. A problem were encountered straight away involved the pathfinding system. The pathfinding system could only handle one search at any time and when multiple agents tried to obtain a route it would cause the application to crash, as a search already being performed would have its goal and target positions changed to what another agent had entered. To fix this problem we make a new function to handle the pathfinding, and make all other functions private so that this was the only access point to the pathfinding system. The system would return a false value if the system was already running a search for another agent. The agent would remain in the new target state until it was that agents' turn to use the pathfinding system. Table 9 shows the results summary collected from this testing stage.

Table 9
Stage 1 testing results summary

Agents	10 seconds	30 seconds	60 seconds	90 seconds	120 seconds
10	164.76	161.56	159.7033	158.1867	157.065
25	104.42	98.32	96.51	95.71555	95.21053
50	63.6	57.96667	56.39333	55.74889	55.24666
100	35.78	30.41333	29.11667	28.72889	28.51167

Table 10
Number of agents and frame rates in stage 2

Agents	10 sec	30 sec	60 sec	90 sec	120 sec
10	170.56	171.6	170.35	169.7067	168.57
25	105	102.1533	100.9833	100.5355	100.0522
50	68.12	62.84	61.38	60.90222	60.565
100	39.4	35.10667	34.11	33.77778	33.66167

7.6. Optimization

To improve the frame rates achieved in stage one testing, we optimize how the application to overcome bottleneck problems. The only places that need optimizing to improve the frame rate are the update and render functions. The quicker these are completed, the faster the application will run. Optimizing the initialization function would only help to speed up the loading time of the application, but not the frame rate.

To reduce the number of agents that each agent is tested against, we have changed how this data is collected. The scene is split into 16 scene nodes, and only the agents that reside in his current scene node will be tested against. This means that fewer tests need to be conducted in getting the neighbour information. This should increase the frame rate as less work now needs to be done. A sorting system was required to sort the agents into the scene nodes, but this also helps to improve the frame rate. If a ground tile is not rendered, then all the agents that reside in those tiles do not need to be rendered either. In the testing for the final project, all 16 ground tiles are rendered. In the testing no improvement will be seen from this, but when the camera is free to move around the scene, it should help to increase the frame rate.

The XNA CLR (Common Language Runtime) handles data variables differently on the Xbox 360 console. Hargreaves [8] described a way to optimise the XNA variables by using inline functions. He used the Vector3 as his example, as this required 3 floats in one of its constructors. It improves the time to create a Vector3 instance, by assigning each of the X, Y and Z values individually.

Passing data into functions with the "ref" keyword will mean that the value with not be copied, which will speed up processing time, but because this is a reference of the value, it can be modified in the function. We use reference parameters in function wherever a larger data type is used, such as when the view and projection matrices are passed into the render avatar function.

The timing system for the animation manager only requires one part of the "GameTime" class, which is a "TimeSpan" variable. This class now only takes a reference to a "TimeSpan" variable.

Granberg [7] suggested that characters using the low resolution LOD should not be playing any animations. We add an additional default animation to the animation manager that does not update, i.e. a static pose, which is used by agents in the low LOD grade. Each grade of LOD has a different set of lighting settings. Agents near the camera viewpoint will be more lit than those in the other two LODs (median, low). The agents in the low LOD are unlit; saving time that would be needed for the lighting calculations. The number of polygons rendered by each avatar cannot be changed as this is all done in the XNA framework, which we have no access.

Updating the world manager for the ground tile and building LOD settings, if input is detected. As the world uses the camera position for updating the LOD on the environment, if no input is detected, then the camera position has not changed and the current LOD levels for the environment would not be changed.

7.7. Stage 2 testing

In stage 2, we optimize the simulator using the recommendations found in the previous sections. The testing is conducted under the same testing conditions. Table 10 shows the summary results collected from this testing stage.

The improvements and optimizations made to the application before stage two testing resulted in an improved frame rate in all areas. The best improvement

Table 11
Percentage increase between stage 1 and 2

Agents	10 sec	30 sec	60 sec	90 sec	120 sec
10	3.52%	6.21%	6.67%	7.28%	7.32%
25	0.56%	3.90%	4.64%	5.04%	5.09%
50	7.11%	8.41%	8.84%	9.24%	9.63%
100	10.12%	15.43%	17.15%	17.57%	18.06%

was achieved in the 100 agent tests, where the average maximum increase in frames was 18.06% which was 5.15 extra frames per second. Table 11 shows the percentage improvements stage 2 had over stage 1. The tests with 25 agents resulted in an improvement less than in the tests with only 10 agents. All tests were conducted in the same testing environment.

Stage two had a feature that sorted the agents into scene nodes. When the neighbour data for an agent was requested, only the agents in the current agents' node would be tested against. In the tests with a large number of agents (50 and 100), the agents did not need to test against all other agents in the scene. This is a likely reason why these tests had a better percentage increase on stage one testing. Stage two used the agent scene node data for rendering as well. But because all scene nodes were visible in the tests, none were culled, so all agents were rendered. In a scene with only a few scene nodes visible, this would most likely improve the frame rate even more.

8. Conclusion

Most crowd simulators use Reynolds [14] rules as a starting point for getting a basic crowd working. These rules are valuable, as these give any crowd a cheap system of order. Without these rules, we would have had to create a system to handle collision avoidance between agents. The agents in our crowd simulation also use a two-part goal system for moving around [16]. The main goal is the intended target that the agent is heading towards, and the other is a sub-goal. The agents use a finite-state machine, and the number of states is kept low. The transitions between states are simple calculations. Keeping track of a large number of agents using a FSM becomes difficult as different agents may be in different states at any time. The best pathfinding algorithm for a crowd simulator is the A* search algorithm. This produced the best results out of the other algorithms tested. The breadth-first search algorithm did give the same route cost as the A* algorithm, but as the A* algorithm was faster, it is adopted.

We used the Xbox 360 console for the ability to do testing which would be uniform and not affected by other processes running in the background. Developing a version on the Xbox 360 with the XNA framework is slow. When we placed a breakpoint on the computer it takes a few seconds before the application hits it. It would have been easier to create a version on the PC first using temporary models instead of the avatars. Using the Xbox avatars was only consider, when research was being done into character creation. The Xbox avatars have some disadvantages: firstly, they can only be rendered on the Xbox 360 console; secondly, they require a content pipeline extension to play custom animations. The advantage they offer is that they are generated at run-time and do not require any stored assets. Only a few avatar models should be created for all the agents in the scene. There is no need for every agent in the scene to have their own model, which would cause memory to be exhausted very fast.

The XNA framework offers no creation tools for level of detail for any geometry. This would have helped in this project as the different versions of the geometry had to be created 3 times (low, medium and high resolution). The ground tiles and building objects were generated in code. For future development with more decent building geometries, a 3D creation tool should be used. Singleton manager classes are used to stop assets being loaded multiple times. Agents have no local copies of assets that are used by other agents. These assets are stored in an asset manager class. Newly created agents in the scene are distributed around the scene and not all placed in the same general location. We optimize the code at the final stage of the project, e.g. data types passed into functions should be as small as possible and we pass them by reference parameters.

To achieve a healthy frame rate of 60 FPS, it would be best to limit the maximum number of agents in the scene to 50. The testing we conducted did render all scene node, so all agents were visible. But in a scene with the camera moving around, we believe that a greater number of agents can be used. 100 agents in the scene produced an average FPS of 33. With the scene node culling, it is most likely that the scene could achieve a better frame rate. From this we suggest that the maximum number of agents to be in a scene at any time be 100, as long as not all scene nodes are rendered.

We believe that our crowd simulator adopts the best performing algorithms of crowd behaviours, pathfinding, collision avoidance, character creation, and level of details on the Xbox platform.

8.1. Future work

Future development should consider rebuilding the project with an Xbox 360 SDK. Using the SDK would allow better access to the avatar rendering system. The SDK is for creating full games, where better memory and processing power can be used. XNA is more designed for the creation of hobby games that do not require too much of the main system. We believe that a version of the application created on an Xbox 360 SDK would give better results than those achieve in this project.

The pathfinding system should consider using a navigation mesh instead of a graph. This would require less memory to be used and would create paths that were more direct, and not just following a grid layout. Creating a new asset content pipeline extension to handle the creation of the different LOD versions of models loaded. This is currently missing in the XNA framework, but a pipeline extension applied to models loaded, could create the different version automatically. Threads should be used to handle large tasks, such as pathfinding. Concurrent execution would help to avoid bottlenecking. Creating the environment texture used for the scene at run-time is also suggested.

References

[1] T. Akenine-Möller, E. Haines and N. Hoffman, Acceleration Algorithms, in: *Real-Time Rendering*, (3rd ed.), Wellesley, MA, USA: A K Peters Ltd, 2009, p. 683.

[2] R. Berggren, Simulating Crowd Behaviour in Computer Games'. BSc dissertation. Luleå University of Technology, 2005.

[3] R. Brooks, A Robust Layered Control System for a Mobile Robot, *IEEE Journal of Robotics and Automation* **2** (1985), 14–23.

[4] M. Buckland, The Secret Life of Graphs, in: *Programming Game AI by Example*, Plano, TX, USA: Woodware Publishing, Inc, 2005, pp. 209–204.

[5] C. Foudil, D. Noureddine, C. Sanza and Y. Duthen, Path Finding and Collision Avoidance in Crowd Simulation, *Journal of Computing and Information Technology* **17**(3) (2009), 217–228.

[6] C. Granberg, Creating the Terrain, in: Programming an RTS Game with Direct3D. Boston, MA, USA: Charles River Media, 2007.

[7] C. Granberg, Crowd Simulation, in: Character Animation with Direct3D, Boston, MA, USA: Charles River Media, 2009.

[8] S. Hargreaves, 2007, Inline those vector constructors, [online]. Available at: blogs.msdn.com/shawnhar/archive/2007/01/02/inline-those-vector-constructors.aspx (Accessed 11th January 2010).

[9] P.E. Hart, N.J. Nilsson and B. Raphael, A Formal Basis for the Heuristic Determination of Minimum Cost Paths, *IEEE Transactions on Systems Science and Cybernetics SSC4* **4**(2) (1968), 100–107.

[10] H. Hoppe, 'Progressive Meshes'. SIGGRAPH: ACM Special Interest Group on Computer Graphics and Interactive Techniques 23. New Orleans Louisiana USA, 4–9 August 1996. New York, NY, USA: ACM, 1996, pp. 99–108.

[11] Z. Kasap and N. Magnenat-Thalmann, Intelligent virtual humans with autonomy and personality: State-of-the-art, *In Intelligent Decision Technologies* **1** (2007), 3–15. IOS Press.

[12] M. McShaffry, *Game Coding Complete*, (3rd ed.), Boston, MA, USA: Delmar, (2009).

[13] I. Millington and J. Funge, *Artificial Intelligent for Games*, (2nd ed.), Burlington, MA, USA: Morgan Kaufmann, (2009).

[14] C. Reynolds, Flocks, Herds, and Schools: A Distributed Behavioural Model, SIGGRAPH: ACM Special Interest Group on Computer Graphics and Interactive Techniques 14. Anaheim California USA, 27–31 July 1987. New York, NY, USA: ACM, 1987.

[15] K. Sornum, Y. Liang, W. Cai, M. Low and S. Zhou, 3D Visualization and Animation of Crowd Simulation Using a Game Engine, Nanyang Technological University, Singapore, 2008.

[16] M. Sung, M. Gleicher and S. Chenney, Scalable behaviours for crowd simulation, Eurographics 2004. Grenoble France, 30 August–3 September 2004. Grenoble: Eurographics, 2004.

[17] F. Tecchia, Behaviour Simulator (ABS): A Platform for Urban Behaviour Development, University College London, 2001.

[18] J. Therien and S. Bernard, Taming the Mob: Creating Believable Crowds in Assassin's Creed. Game Developers Conference, San Francisco, CA, USA: Ubisoft, 2008.

[19] S. Van den Hurk and I. Watson, A Multi-Layered Flocking System for Crowd Simulation, in the Proceedings of the 3rd Annual International Conference on Computer Games, Multimedia and Allied Technology (CGAT 2010), E. Prakash ed., 184–191, Singapore Management University, Singapore, 6–7 April 2010. APTF: Singapore, 2010.

[20] XNA reators' club. [Online], http://creators.xna.com/en-GB/sample/customavataranimation.